Hochschule Rhein-Waal

HOCHSCHULE
RHEIN-WAAL
Rhine-Waal University
of Applied Sciences

and

Fluxana GmbH & Co. KG

FLUXANA®
XRF Application Solutions

Robotic arm Automation – using ROS (Robot Operating System)

---

05 June 2023

Internship Report

Abdelrahman Mostafa (29528)

Bachelor of Mechatronic Systems Engineering

Supervised by:

Dr. Rainer Schramm, Fluxana GmbH
Prof. Dr. Ronny Hartanto, HSRW

ABSTRACT

This internship report presents the implementation of the Niryo Ned2 robot in conjunction with the Fluxana library to accomplish a powder dosage task. By leveraging the capabilities of ROS (Robotic Operating System) and Moveit (Motion Planning Framework), which are seamlessly integrated with the Fluxana cobot library, the internship aimed to address the need for an efficient and precise powder dosage process.

Motivation:

The motivation behind this internship project stems from the growing demand for automated and accurate powder dosage tasks. Traditional manual methods are often time-consuming, labour-intensive, and prone to human error. By employing robotics technology and leveraging the potential of collaborative robots (cobots), such as the Niryo Ned2, the aim is to enhance the efficiency and precision of powder dosage operations. The integration of ROS, Moveit, and the Fluxana library provides a powerful platform to achieve these objectives.

Problem:

The challenge at hand involves seamlessly integrating the Niryo Ned2 robot with the Fluxana library and optimizing its functionalities to perform the powder dosage task effectively. This requires in-depth knowledge of ROS and Moveit frameworks, as well as the ability to expand the Fluxana library's capabilities to accommodate the specific requirements of the Niryo Ned2 robot. Additionally, the performance and validation of the powder dosage task need to be thoroughly assessed to ensure its accuracy and reliability.

Through this internship, the primary learning objectives were to gain comprehensive knowledge of integrating the Niryo Ned2 robot with ROS and Moveit, expand the functionalities of the Fluxana library to suit the Niryo Ned2 robot's requirements, and successfully execute and validate a powder dosage task.

By accomplishing these objectives, this internship report contributes to the advancement of automation in powder dosage processes, offering potential improvements in efficiency, accuracy, and repeatability.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1   INTRODUCTION

The field of robotics has witnessed remarkable progress in recent years, revolutionizing various industrial processes and offering solutions to complex tasks. One crucial area of focus involves the programming and automation of robotic arms, which play a pivotal role in executing precise and repetitive operations across multiple industries. This report presents the outcomes and discoveries from an internship project that aimed to program and automate the Niryo 2 robotic arm utilizing the ROS1 (Robot Operating System) framework.

The Niryo 2 robotic arm provides a flexible and cost-effective solution for automation tasks, thanks to its modular design and user-friendly control interfaces. This internship project focused on harnessing the capabilities of the ROS1 ecosystem, particularly the MoveIt motion planning framework, to enhance the motion planning and manipulation capabilities of the Niryo 2 arm. By leveraging the powerful tools and libraries offered by ROS1, the objective was to optimize trajectory generation, improve collision avoidance, and enable efficient object manipulation in diverse scenarios.

This report provides a comprehensive overview of the internship project, outlining the objectives, methodology, and significant findings. It offers insights into the research question, the experimental setup, and the implementation of advanced motion planning algorithms using the MoveIt package. Furthermore, it addresses the challenges encountered during the project and discusses the potential implications and future directions of the research.

The research conducted in this internship project contributes to the broader field of robotics and automation by exploring the application of ROS1 and MoveIt in programming and automating the Niryo 2 robotic arm. The findings and insights gained from this project serve as a foundation for further advancements in motion planning, manipulation, and human-robot interaction.

In summary, this report provides a comprehensive account of the internship project's progress, documenting the achieved milestones, lessons learned, and the potential impact of the research. It serves as a valuable resource for researchers, practitioners, and enthusiasts interested in programming and automating robotic arms using ROS1 and the Niryo 2 platform. The outcomes of this project pave the way for continued advancements in robotic arm automation, enhancing the efficiency and effectiveness of industrial processes.

Research Question: To what extent can the integration of the Niryo 2 robotic arm with the ROS1 framework, incorporating the MoveIt motion planning framework, optimize motion planning, manipulation, and automation for efficient execution of a powder dosage task?

Figure 1: Niryo Ned 2 Robot

2   RELATED WORKS

This chapter of our research paper provides an extensive review of studies focusing on the integration of robotic arms with motion planning frameworks, particularly within the context of the ROS1 framework. We explore previous research that has examined the optimization of motion planning, manipulation, and automation capabilities in various applications.

One relevant study by (St-Onge & Herath, 2022) investigated the integration of a robotic arm with ROS1 and the MoveIt framework for precise object manipulation tasks in manufacturing. Their work emphasized the use of advanced motion planning algorithms and collision avoidance techniques to improve the efficiency and accuracy of the robotic arm's movements.

Additionally, the research conducted by Chand (2022) focused on the development of a novel Matlab controller for the Niryo Ned Robot. The Ned robot, an open-source technology robotic arm designed for education, research, and vocational training, provides Niryo Studio as a configuration and basic program implementation tool. However, the current control capabilities of the Ned robot through Matlab are limited. Chand addressed this limitation by developing a Matlab controller, which offers the advantage of leveraging various toolboxes for machine learning, control, and image processing, making it suitable for research and development purposes.

Chand's research paper presented the implementation and evaluation of the developed Matlab controller on a real Niryo Ned Robot. Four target coordinate locations representing the edges of the robot's workspace were utilized for evaluation. The results demonstrated the functionality and effectiveness of the Matlab controller in controlling the Niryo Ned Robot. This study aligns with our research paper as it also explores the control aspects of the Niryo Ned Robot. However, while Chand's work primarily focused on the development of the Matlab controller, our research aims to extend the capabilities of the Niryo Ned2 robot by integrating it with the ROS1 framework and MoveIt using Python user interface to control. By integrating the insights gained from Chand's study and our own research, we can further enhance the control and automation functionalities of the Niryo Ned2 robot, specifically for powder dosage tasks.

While these studies have made significant contributions to the field, they primarily focused on specific applications or aspects of robotic arm integration. In our research, we aim to build upon their findings by specifically addressing the optimization of motion planning, manipulation, and automation for a powder dosage task using the Niryo 2 robotic arm. By examining the gaps and limitations identified in previous studies, we intend to extend the existing knowledge and contribute to the advancement of efficient powder dosage tasks in a manufacturing setting.

3   SUMMARY OF INTERNSHIP ACTIVITIES

In this chapter, we present a comprehensive Summary and overview of the key activities undertaken during the internship program focused on programming and automating the Niryo 2 robotic arm using the ROS1 framework. The chapter highlights the learning process, and milestones achieved throughout the internship, covering a range of essential topics.

The chapter begins with Activity 1: Getting familiar with Linux and Ubuntu. This activity focuses on familiarizing oneself with the Linux operating system and Ubuntu distribution, which serve as the foundation for ROS1 development. It includes exploring the command-line interface, file system navigation, package management, and basic shell scripting. A solid understanding of Linux and Ubuntu is crucial for effectively working with ROS1.

Activity 2: Getting to Know ROS1 forms the next phase of the internship. This activity delves into the fundamental concepts of ROS1, including its architecture, communication mechanisms, and core components. It explores topics such as nodes, topics, services, and actions, which are the building blocks of ROS1-based applications. A deep understanding of ROS1 is vital for leveraging its capabilities and developing robust and efficient robotic systems.

Activity 3: Building an Automation Program using the Moveit framework and tkinter for the GUI represents a significant milestone in the internship. This activity focuses on integrating the Moveit framework, a powerful motion planning library, with a graphical user interface (GUI) developed using the tkinter library. The goal is to create an intuitive and user-friendly interface for controlling the Niryo 2 robotic arm. The activity involves designing and implementing motion planning algorithms, handling user inputs, visualizing the robot's state, and executing motion trajectories.

Activity 4: Making a General-Purpose Python Library with Moveit aims to extend the capabilities of the Moveit framework by developing a reusable and optimized Python library. This library provides a set of high-level functions and utilities that simplify common tasks in robotic motion planning and automation. The activity involves designing the library's API, implementing key functionalities, and optimizing the codebase for improved performance and usability. The resulting library can be utilized in various robotics projects, enhancing productivity, and promoting code reusability.

Throughout the following chapters, each activity is discussed in detail, encompassing the objectives, methodologies, challenges faced, and the corresponding outcomes. Practical examples, code snippets, and graphical representations are provided to illustrate the implementation process. The chapter serves as a comprehensive record of the internship's progression, showcasing the acquisition of technical skills, problem-solving abilities, and the application of theoretical knowledge in real-world scenarios.

4   LINUX OS AND UBUNTU

Linux is a free, open-source operating system that includes several utilities that will significantly simplify your life as a robot programmer. And as will be shown in the next chapter, ROS (Robot Operating System) is based on a Linux system. All commands and concepts explained here are taken from the Linux tutorial made by the University of Surrey.[1]

4.1   DIRECTORY STRUCTURE

The files are organized collectively within the directory structure. The file system is structured hierarchically, resembling an upside-down tree. The highest level of the hierarchy is commonly referred to as the root, represented by a forward slash (/).
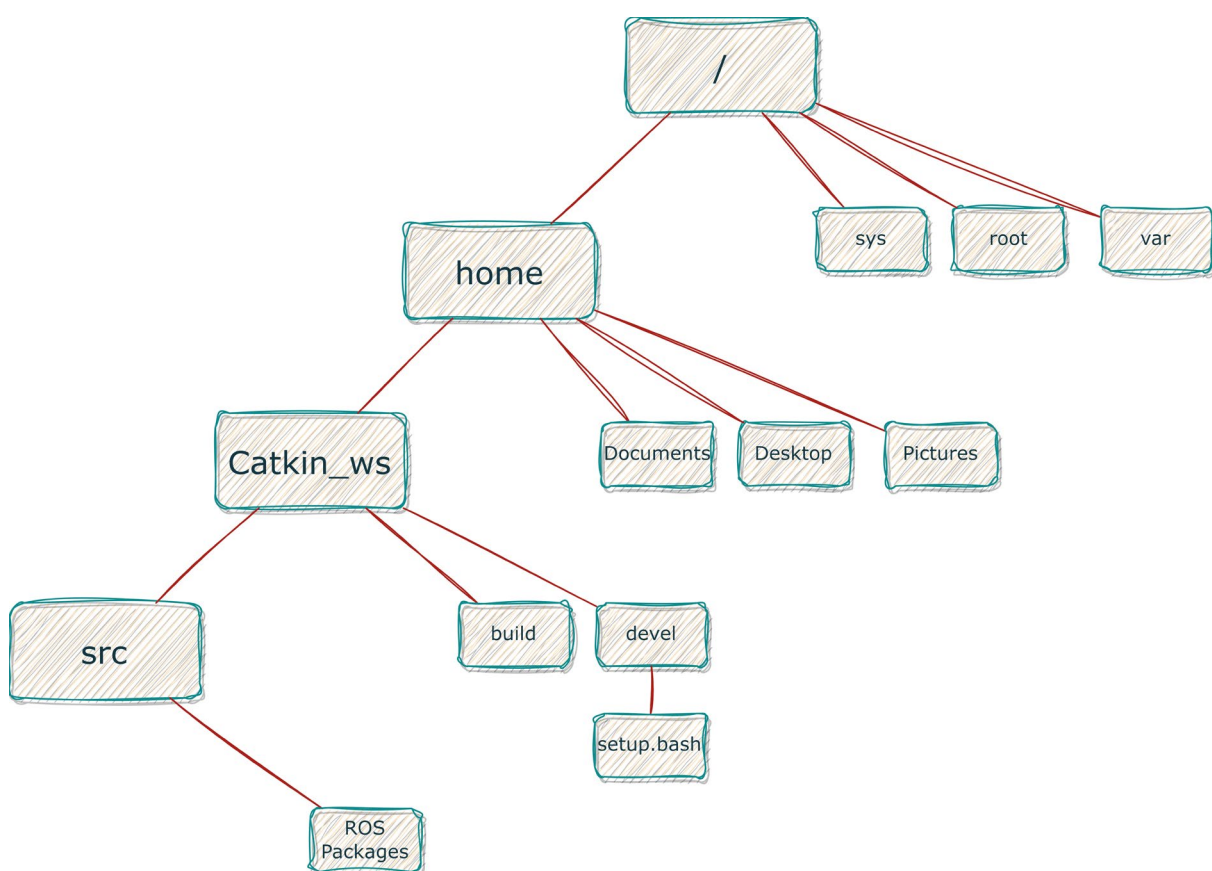
Figure 2: Linux System Directory Structure

The diagram presented in *Figure 2* illustrates the directory organization within our Linux system. It begins with the root folder (/) and concludes at the /src folder, where all the ROS packages are located.

---

[1] (Stonebank, 2000)

4.2   IMPORTANT LINUX COMMANDS

Table 1 below provides a comprehensive list of essential Linux commands. These commands will serve as a valuable reference for me throughout the duration of my internship[2].

Table 1: Important Linux Command lines

Command	Meaning
ls	list files and directories
ls -a	list all files and directories
mkdir	make a directory
cd *directory*	change to named directory
cd	change to home-directory
cd ~	change to home-directory
cd ..	change to parent directory
pwd	display the path of the current directory
cp file1 file2	copy file1 and call it file2
mv file1 file2	move or rename file1 to file2
rm file	remove a file
rmdir directory	remove a directory
cat file	display a file
less file	display a file a page at a time
head file	display the first few lines of a file
tail file	display the last few lines of a file
grep 'keyword' file	search a file for keywords

---

[2] (Stonebank, 2000)

5   GETTING TO KNOW ROS1

ROS, short for Robot Operating System, deviates from the conventional concept of an operating system that primarily focuses on process management and scheduling. Instead, ROS serves as a structured communication layer that operates above the host operating systems within a diverse compute cluster.[3]

Nodes in ROS are computational processes that operate at a granular level. The design of ROS emphasizes modularity, typically consisting of numerous interconnected nodes or software modules. The term "node" is used to describe these software modules, as it visualizes the runtime behaviour of ROS-based systems as a graph representation. The communication between nodes occurs through message passing, enabling effective peer-to-peer interactions.

Messages in ROS are structured data entities with strict data types. They support standard primitive types such as integers, floating-point numbers, and Booleans, as well as arrays of primitive types and constants. Messages can be composed of other messages and nested arrays of messages to any arbitrary depth. Nodes in ROS publish messages by associating them with specific **topics**, which are identified by simple strings like "odometry" or "map." Conversely, nodes interested in particular data subscribe to relevant topics. Multiple **publishers** and **subscribers** can exist for a single topic, and a node can publish and/or subscribe to multiple topics. Publishers and subscribers generally have no knowledge of each other's presence.

Although the topic-based publish-subscribe model in ROS offers flexibility in communication, its "broadcast" routing mechanism is not suitable for synchronous transactions that require a more streamlined node design. In ROS, this is addressed through a concept called services. A **service** is identified by a string name and consists of a pair of strictly typed messages: one for the request and one for the response. This concept is analogous to web services, which are defined by URIs and have well-defined types for request and response documents. It is important to note that unlike topics, only one node can advertise a service with a specific name. For instance, there can only be one service named "classify image," just as there can only be one web service associated with a particular URI.
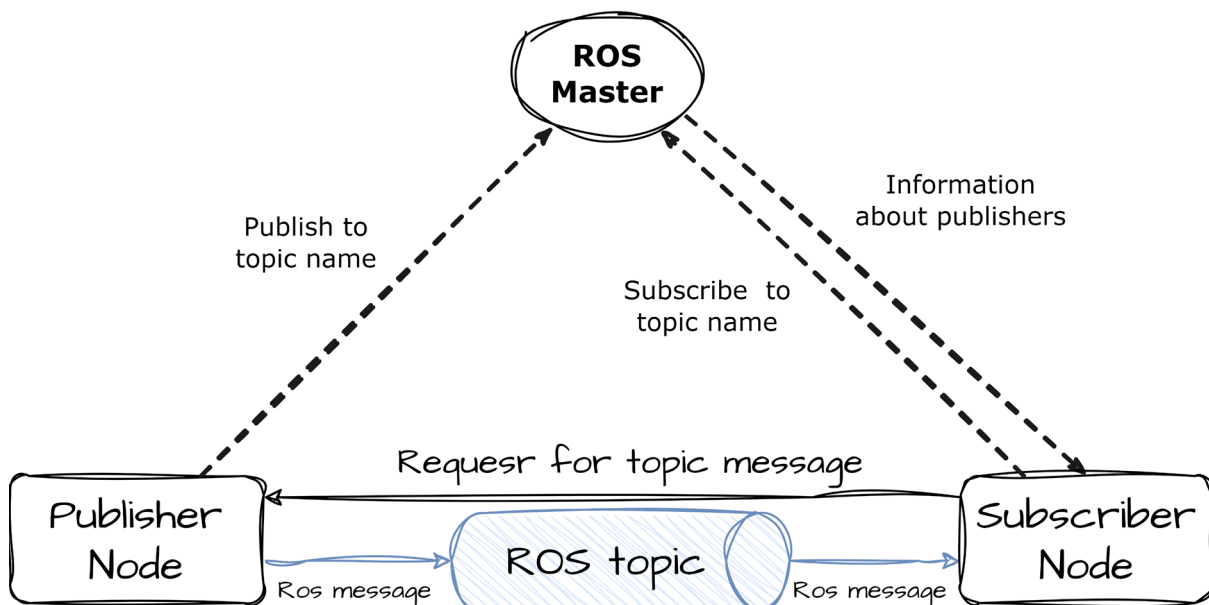

Figure 3: Communication in ROS

---

[3] (Quigley et al., 2009)

5.1   MOVEIT

MoveIt! serves as the primary software framework within ROS for motion planning and mobile manipulation. It has achieved successful integration with various robots, including the PR2, Robonaut, and DARPA's Atlas robot. MoveIt! is predominantly implemented in C++, but it also provides Python bindings for higher-level scripting. In line with the principles advocated for robotics, MoveIt! ensures software reuse by establishing a clear distinction between core functionality and framework-dependent aspects, such as component communication. By default, MoveIt! leverages the core ROS build and messaging systems. To facilitate easy component interchangeability, MoveIt! employs plugins for most of its functionalities. This includes motion planning plugins (currently utilizing OMPL), collision detection (currently utilizing the Fast Collision Library (FCL)), and kinematics plugins (currently utilizing the OROCOS Kinematics and Dynamics Library (KDL)) for generic arm forward and inverse kinematics, along with custom plugins). Section 4.3 further discusses the flexibility of modifying these default planning components[4]. MoveIt! primarily targets manipulation (and mobile manipulation) applications across industrial, commercial, and research environments. For a more comprehensive overview of MoveIt![5].
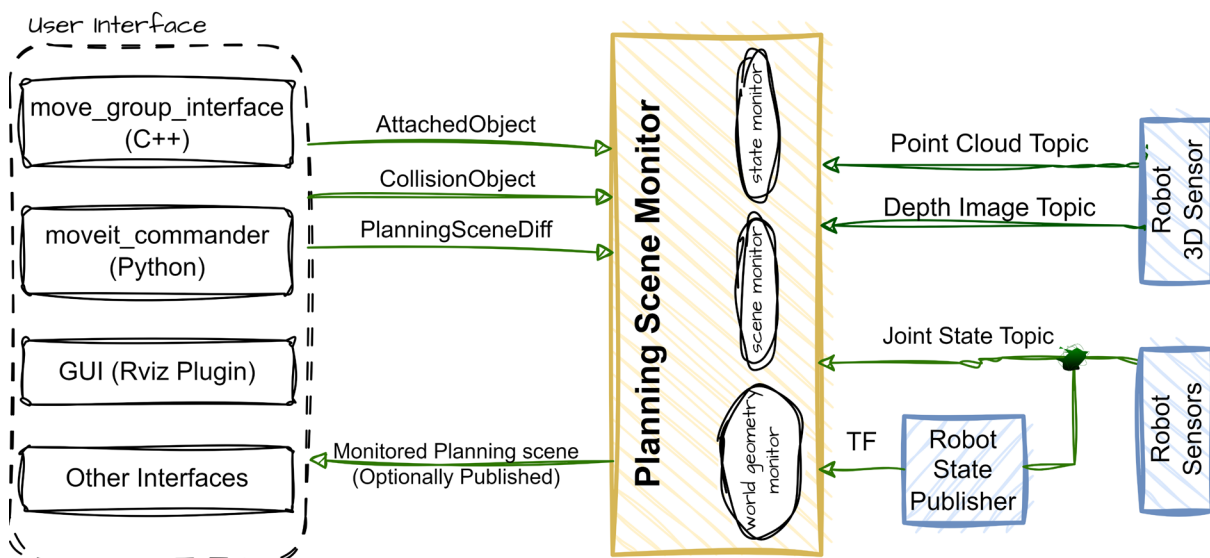

Figure 4: The Planning Scene in MoveIt!

As shown in *Figure 4* above, the planning scene serves as a representation of the robot's surrounding environment and retains information about the robot's current state. It is managed by the planning scene monitor within the move group node. The planning scene monitor actively listens to three types of information:

1. State Information: This includes updates on the joint states of the robot, which are obtained from the joint_states topic.
2. Sensor Information: The planning scene monitor gathers data from sensors through the world geometry monitor, which is further described below.
3. World Geometry Information: Users can input world geometry information through the planning_scene topic, which is treated as a planning scene diff and used to update the planning scene accordingly.

---

[4] (S. Chitta, Sucan, & Cousins, 2012)
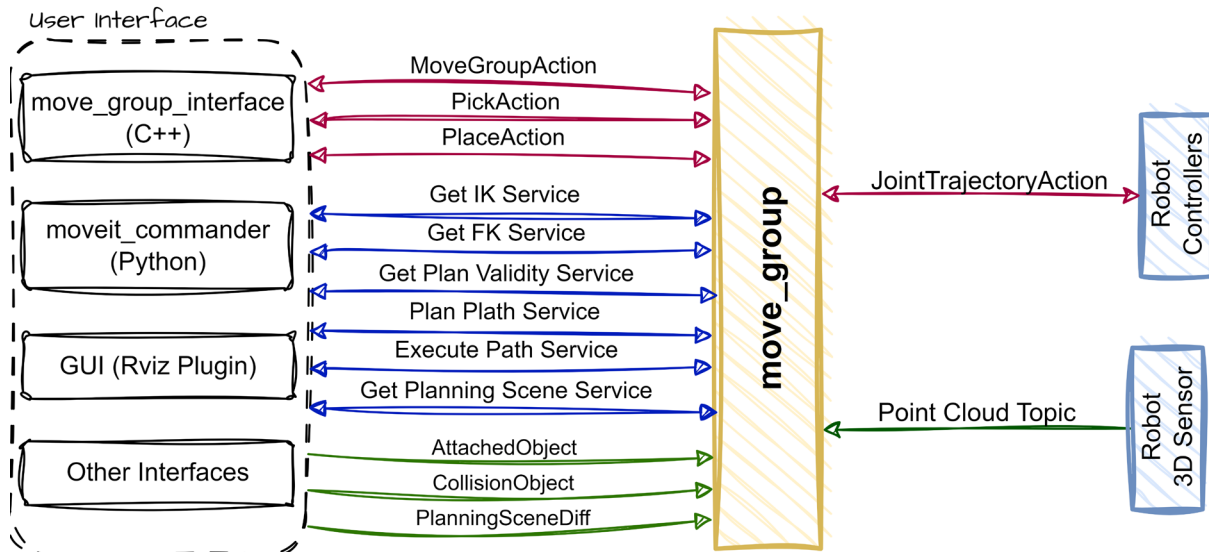
[5] (I. A. S. u. a. S. Chitta, 2013)

Figure 5: Move_group Node in MoveIt!

Figure 5 depicts the move_group node, which serves as an integrator, bringing together various components to offer a set of ROS actions and services for user interaction. Users can access these actions and services through three different interfaces[6]:

- **C++ Interface**: The move_group_interface package provides a straightforward C++ interface for easy setup and utilization of move_group.
- **Python Interface**: The moveit_commander package enables users to access move_group functionalities through Python.
- **GUI Interface**: Users can also employ the Motion Planning plugin in Rviz, the ROS visualizer, to interact with move_group through a graphical user interface.

To configure move_group, the ROS param server is utilized to obtain essential information. This includes:

1. **URDF**: move_group retrieves the URDF (Unified Robot Description Format) for the robot by searching for the robot_description parameter on the ROS param server.
2. **SRDF**: The robot_description_semantic parameter on the ROS param server contains the SRDF (Semantic Robot Description Format) for the robot. The SRDF is typically generated by users using the MoveIt Setup Assistant.
3. **MoveIt Configuration**: Specific configuration details for MoveIt, such as joint limits, kinematics, motion planning, and perception information, are stored on the ROS param server. Configuration files for these components are automatically generated by the MoveIt Setup Assistant and stored in the config directory of the corresponding MoveIt configuration package for the specific robot.

---

[6] (I. A. S. u. a. S. Chitta, 2013)

6   1ST AUTOMATION PYTHON PROGRAM USING MOVEIT_COMMANDER & TKINTER

Activity 3 represents a significant milestone in the internship program, focusing on the development of an automation program for the Niryo 2 robotic arm using the Moveit framework and the tkinter package for creating a graphical user interface (GUI). This activity combines the power of motion planning provided by Moveit with the versatility of tkinter to create an intuitive and interactive control interface.

The core component of this activity is the utilization of the Moveit_commander library, a purpose-built Python library designed to facilitate seamless interaction with the Moveit framework. This library provides a high-level interface for controlling the robotic arm, enabling the execution of complex motion trajectories, joint configurations, and end-effector poses. By leveraging the Moveit_commander library, the automation program can communicate with the Niryo 2 robotic arm, translating user commands into precise and coordinated movements[7].

Furthermore, this activity incorporates the tkinter package, a popular Python library for creating graphical user interfaces. Tkinter provides a rich set of tools and widgets that enable the creation of visually appealing and user-friendly GUIs. With tkinter, the automation program can present a comprehensive control panel to the user, offering intuitive controls, real-time visual feedback, and status updates. The GUI enhances the user experience by providing a seamless and interactive means of commanding the Niryo 2 robotic arm[8].

Throughout the development process, the activity covers various aspects, including designing the GUI layout, implementing event handling mechanisms, integrating the Moveit framework with the GUI, and ensuring synchronization between the interface and the robot's actions. Attention is given to designing an ergonomic and intuitive interface that enables users to perform tasks such as motion planning, trajectory execution, and monitoring of the robot's state effortlessly.

By combining the Moveit framework and tkinter, this activity demonstrates the seamless integration of motion planning capabilities with a user-friendly interface. The resulting automation program empowers users to control the Niryo 2 robotic arm efficiently, perform complex manipulation tasks, and visualize the robot's movements in real-time.

In the subsequent sections, we will explore the implementation details of Activity 3, including the design considerations, code structure, and the interplay between the Moveit_commander library and tkinter. The discussion will provide valuable insights into the development process and highlight the contributions of Moveit and tkinter in creating an effective and interactive automation program for the Niryo 2 robotic arm.

---

[7] https://docs.ros.org/en/jade/api/moveit_commander/html/namespacemoveit__commander.html
[8] https://docs.python.org/library/tk.html

6.1   DEPENDENCES

The program had many dependences as we did use many libraries, services, and message types.

```python
#!/usr/bin/env python

# General packages
import tkSimpleDialog as simpledialog
import Tkinter as tk
import ttk
from ttkthemes import ThemedTk
from PIL import ImageTk, Image
import tf
import os

# ROS
import rospy

# Services
from niryo_robot_arm_commander.srv import GetFK, GetFKRequest, GetIK,
GetIKRequest
from tools_interface.srv import ToolCommand, ToolCommandRequest
from niryo_robot_msgs.srv import SetBool, SetBoolRequest, SetInt,
SetIntRequest, Trigger

# Messages
from geometry_msgs.msg import Pose
from sensor_msgs.msg import JointState
from niryo_robot_msgs.msg import RobotState
import moveit_msgs.msg

import actionlib

# moveit_library
import moveit_commander
```

6.2   GRAPHICAL USER INTERFACE THROUGH TKINTER

In its initial phase, the User interface was limited in functionality, offering basic automation features. The GUI underwent several stages of development, which can be summarized into four key stages. The first and second stages primarily involved the implementation of the following simple code to create the GUI, with not much any organisation at all, as shown in *Figure 6 - 7* below:

```python
import Tkinter as tk


root = tk.Tk()
root.title("FX UR3e Controller")

root.geometry('(Chand, 2022)x(Chand, 2022)'.format(1800,800))
root.state('normal')
left_frame = tk.Frame(root, width=60)
left_frame.pack(fill=tk.Y)
```


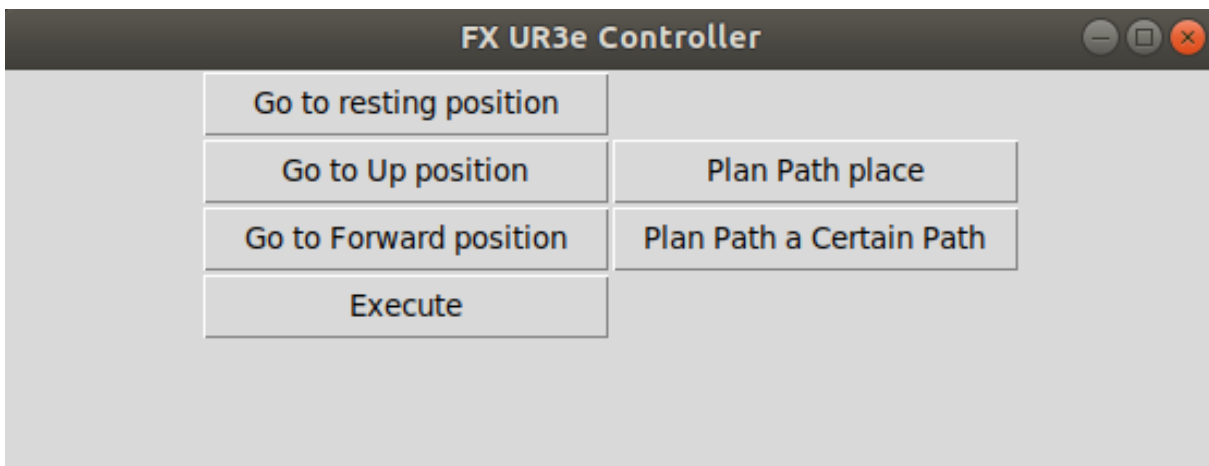**Figure 6: Initial Stage of the GUI**
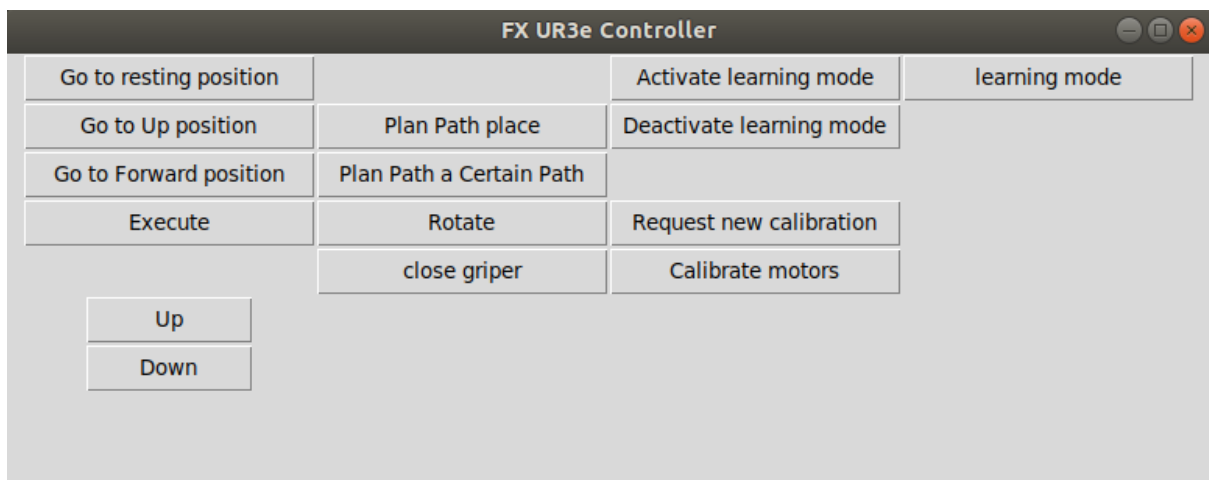
Figure 7: Second Stage of the GUI

Whereas, starting from 3rd stage, the GUI started look more organised, and to have more functionalities, as shown in *Figure 8* below.
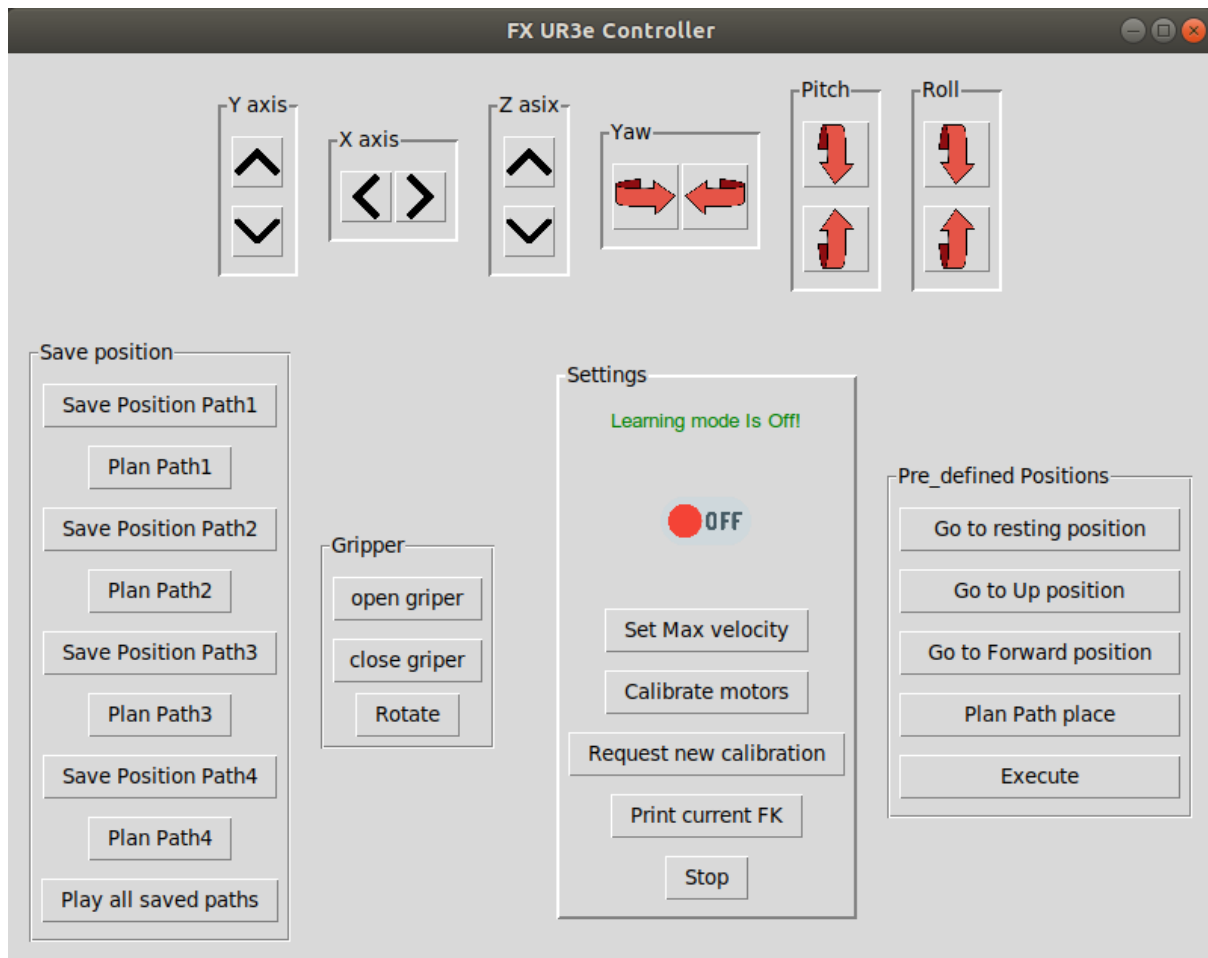
Figure 8: Third Stage of the GUI

Lastly, we reach the final stage, where all the functions, layout, and aesthetically pleasing themes are incorporated into the GUI, resulting in the creation of a comprehensive Graphical User Interface. Refer to *Figure 9* below to visualize the completed GUI.
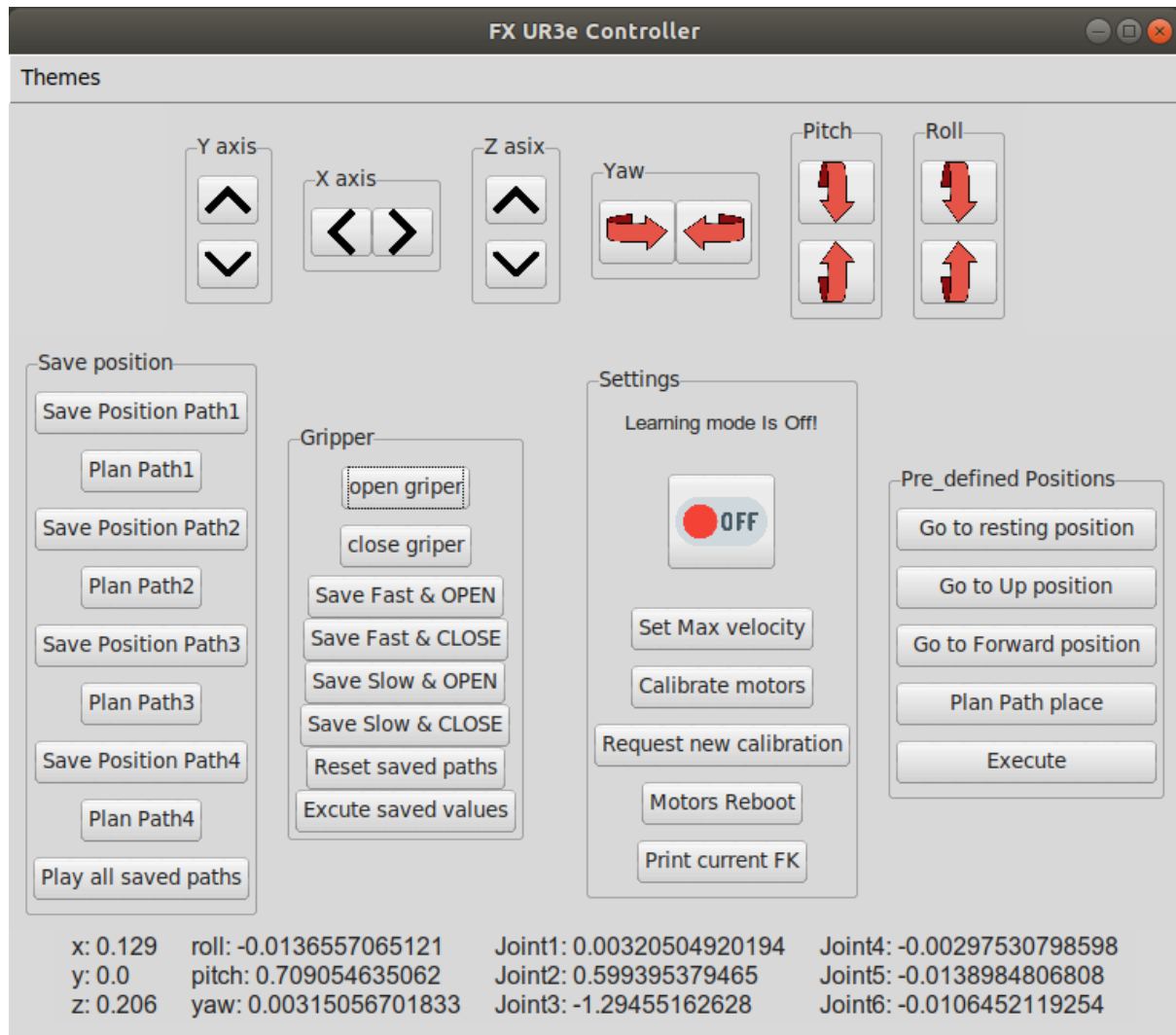
Figure 9: Final Stage in the GUI

Additionally, we offer the flexibility to modify the existing GUI layout by accessing the "Themes" menu located in the top-left corner of the GUI. For detailed insights into the implementation code for this updated GUI, please refer to *Appendix A: Initialization of GUI with tkinter*, located at the end of this report.

6.3   FUNCTIONS & BUTTONS

In this section, I will initially highlight some of the essential functions within the code. However, for a comprehensive understanding of the GUI's construction, including its functions and buttons, please refer to *Appendix B*, which contains the complete code.

6.3.1   MOTOR CALIBRATION

Within this particular function, I invoked the ROS service '/niryo_robot/joints_interface/calibrate_motors' that is provided by Niryo. This service is utilized to initiate motor calibration, and the value of 1 is passed as a parameter to execute the calibration process.

```python
def calibration():
    """Give a value of 1 to the service responsible for auto calibration"""

    rospy.wait_for_service('/niryo_robot/joints_interface/calibrate_motors')
    service =
rospy.ServiceProxy('/niryo_robot/joints_interface/calibrate_motors', SetInt)

    request = SetIntRequest()
    request.value = 1

    response = service(request)

    if response.success:
        rospy.loginfo("Service call succeeded")
    else:
        rospy.logerr("Service call failed")

# Button for Motors calibration.
cal_Btn = ttk.Button(settings_frame,text="Calibrate motors",
command=calibration)
cal_Btn.pack(pady = 5)
```

6.3.2   REQUIEST A NEW CALIBRATION

```python
def request_calibration():
    """Request a new calibration"""

    rospy.wait_for_service('/niryo_robot/joints_interface/request_new_calibrat
ion')
    service =
rospy.ServiceProxy('/niryo_robot/joints_interface/request_new_calibration',
Trigger)

    response = service()
```

6.3.3   GET JOINTS & POSE VALUES

The subsequent pair of functions are responsible for subscribing to and extracting information from two different topics: '/Joint_states' and 'niryo_robot/robot_state'. The purpose of subscribing to the '/Joint_states' topic is to retrieve joint values, while the 'niryo_robot/robot_state' topic is utilized to obtain pose values.

```python
def Get_joints():
    """subscribe to the topic /Joint_states and take the joints' values"""
    msg = rospy.wait_for_message('/joint_states', JointState)
    joints = msg.position

    # return a tuple of 6 value for each joint from 1 till 6.
    return joints

def get_pose():
    """
    @argm: None

    return a RobotState message type, like the following:
position:
  x: ------
  y: ------
  z: ------
rpy:
  roll: ------
  pitch: ------
  yaw: ------
    """
    msg = rospy.wait_for_message('/niryo_robot/robot_state', RobotState)

    return msg

def get_pose_list():
    """
    return the current pose as a list:
    [x, y, z, roll, pitch, yaw]
    """
    value = get_pose()
    p = value.position
    rpy = value.rpy

    return [p.x, p.y, p.z, rpy.roll, rpy.pitch, rpy.yaw]
```

6.3.4   MOVING POSE IN EACH AXIS

Once the current pose and joint values have been acquired, we have the flexibility to manipulate each axis according to our requirements. In the provided code snippet, I have demonstrated an example of moving the z-axis upwards by approximately 20mm. For further insights into the movement of each axis, please refer to the complete code available in *Appendix B*.

```python
def Up_z():
    """Increase the value of Z-axis"""
    global Plan

    FK = get_pose()
    rpy = FK.rpy
    p = FK.position
    pose = Pose()

    z = p.z
    z += 0.02

    pose.position.x = p.x
    pose.position.y = p.y
    pose.position.z = z

    pose.orientation.x, pose.orientation.y, pose.orientation.z,
pose.orientation.w = tf.transformations.quaternion_from_euler(rpy.roll,
rpy.pitch, rpy.yaw)

    arm.set_goal_tolerance(0.01)
    arm.set_pose_target(pose)
    Plan = arm.plan()

    Update_position()

# To excute the planed rotation.
    arm.execute(Plan,wait=True)

up_img = ImageTk.PhotoImage(Image.open(path + "/up-
arrow.png").resize((30,30)),Image.ANTIALIAS)
down_img = ImageTk.PhotoImage(Image.open(path + "/down-
arrow.png").resize((30,30)),Image.ANTIALIAS)

up_z = ttk.Button(Z_axis_frame, image=up_img, command=Up_z)
up_z.pack(pady = 5)
```

6.3.5   PLAN A PRE-SAVED POSITIONS & EXECUTE IT

```python
def GoHomebtnClick():
    """Plan the the pre-saved position called resting"""

    global Plan
    arm.set_goal_tolerance(0.01)
    arm.set_named_target("resting")
    Plan = arm.plan()

GoHomebtn = ttk.Button(preset_frame,text="Go to resting position",width=20,
command=GoHomebtnClick)
GoHomebtn.pack(pady=5)
```

```python
def ExecutebtnClick():
    """Execute the planed path in the global variable Plan"""

    global Plan
    arm.execute(Plan,wait=True)

    Update_position()
```

7   BUILDING A GENERAL-PURPOSE FLUXANA ROS LIBRARY

The concept of creating a versatile ROS library emerged with the aim of streamlining the development process of a more intricate Graphical User Interface (GUI) and incorporating advanced functionalities into it.

REFERENCES

Chand, P. (2022). *Developing a Matlab Controller for Niryo Ned Robot.* Paper presented at the 2022 1st International Conference on Technology Innovation and Its Applications (ICTIIA).

Chitta, I. A. S. u. a. S. (2013). Moveit! Retrieved from http://moveit.ros.org/

Chitta, S., Sucan, I., & Cousins, S. (2012). Moveit![ros topics]. *IEEE Robotics & Automation Magazine, 19*(1), 18-19.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., . . . Ng, A. Y. (2009). *ROS: an open-source Robot Operating System.* Paper presented at the ICRA workshop on open source software.

St-Onge, D., & Herath, D. (2022). The Robot Operating System (ROS1 &2): Programming Paradigms and Deployment. In *Foundations of Robotics: A Multidisciplinary Approach with Python and ROS* (pp. 105-126): Springer.

Stonebank, M. (2000). Unix tutorial one. Retrieved from http://www.ee.surrey.ac.uk/Teaching/Unix/

APPENDIX

APPENDIX A: INITIALIZATION OF GUI WITH TKINTER, USING PYTHON 2.7

```python
import Tkinter as tk
import ttk
from ttkthemes import ThemedTk

root = ThemedTk()
root.title("FX UR3e Controller")

# Set the size of the GUI window
root.geometry('(Chand, 2022)x{}'.format(800,800))

# Set the initial state of the GUI window
root.state('normal')

# Create a frame to hold the top section of the GUI window
Top_frame = ttk.Frame(root)
Top_frame.pack(fill=tk.Y, side="top")
# Create a frame to hold the middle section of the GUI window
Middle_frame = ttk.Frame(root)
Middle_frame.pack(fill=tk.Y, side="top")
# Create a frame to hold the bottom section of the GUI window
Bottom_frame = ttk.Frame(root)
Bottom_frame.pack(fill=tk.Y, side="top")

# Create a frame to hold the pre-defined positions buttons
preset_frame = ttk.LabelFrame(Middle_frame, text="Pre_defined Positions")
preset_frame.pack(padx=10, pady=10, side="right")

# Create a frame to hold the settings buttons
settings_frame = ttk.LabelFrame(Middle_frame, text="Settings", relief =
tk.RAISED)
settings_frame.pack(padx=10, pady=10, side="right")

# Create a frame to hold the motion buttons
motion_frame = ttk.LabelFrame(Middle_frame, text="Motion")
motion_frame.pack(padx=10, pady=10, side="right")

# Create a frame to hold the save position button
save_frame = ttk.LabelFrame(Middle_frame, text="Save position")
save_frame.pack(padx=10, pady=10, side="left")

# Create a frame to hold the Roll joint controls
roll_frame = ttk.LabelFrame(Top_frame, text="Roll", relief = tk.SUNKEN)
roll_frame.pack(padx=10, pady=10, side="right")

# Create a frame to hold the Pitch joint controls
```

```python
pitch_frame = ttk.LabelFrame(Top_frame, text="Pitch", relief = tk.SUNKEN)
pitch_frame.pack(padx=10, pady=10, side="right")
# Create a frame to hold the Yaw joint controls
yaw_frame = ttk.LabelFrame(Top_frame, text="Yaw", relief = tk.SUNKEN)
yaw_frame.pack(padx=10, pady=10, side="right")
# Create a frame to hold the Z-axis joint controls
Z_axis_frame = ttk.LabelFrame(Top_frame, text="Z asix")
Z_axis_frame.pack(padx=10, pady=10, side="right")
# Create a frame to hold the X-axis joint controls
X_axis_frame = ttk.LabelFrame(Top_frame, text="X axis")
X_axis_frame.pack(padx=10, pady=10, side="right")
# Create a frame to hold the Y-axis joint controls
Y_axis_frame = ttk.LabelFrame(Top_frame, text="Y axis")
Y_axis_frame.pack(padx=10, pady=10, side="right")

# Create a frame to hold the Gripper controls
gripper_frame = ttk.LabelFrame(Middle_frame, text="Gripper")
gripper_frame.pack(padx=10, pady=10, side="left")

style = ttk.Style(root)
style.theme_use("scidsand")

# See included styles
our_themes = root.get_themes()

def changer(theme):
    style.theme_use(theme)

my_menu = tk.Menu(root)
root.config(menu=my_menu)

theme_menu = tk.Menu(my_menu, tearoff=0)
my_menu.add_cascade(label="Themes", menu=theme_menu)

#create Sub menu
for t in our_themes:
    theme_menu.add_command(label=t, command= lambda t=t: changer(t))

is_on = False
# Create Label
my_label = ttk.Label(settings_frame,
    text = "Learning mode Is Off!",
    font = ("Helvetica", 10))

my_label.pack(pady = 5)
```

APPENDIX B: FULL CODE OF THE GUI USED TO PRESENT NIRYO AT HSRW EVENT

```python
#!/usr/bin/env python

# General packages
import tkSimpleDialog as simpledialog
import Tkinter as tk
import ttk
from ttkthemes import ThemedTk
from PIL import ImageTk, Image
import tf
import os

# Services
from niryo_robot_arm_commander.srv import GetFK, GetFKRequest, GetIK,
GetIKRequest
from tools_interface.srv import ToolCommand, ToolCommandRequest
from niryo_robot_msgs.srv import SetBool, SetBoolRequest, SetInt,
SetIntRequest, Trigger

# Messages
from geometry_msgs.msg import Pose
from sensor_msgs.msg import JointState
from niryo_robot_msgs.msg import RobotState
import moveit_msgs.msg

import moveit_commander

rospy.init_node('controller')

path = os.path.expanduser('~/Documents')
path = path + "/images"

robot = moveit_commander.RobotCommander()
scene = moveit_commander.PlanningSceneInterface()
arm = moveit_commander.move_group.MoveGroupCommander("arm")

root = ThemedTk()
root.title("FX UR3e Controller")
global Plan
global Path1; Path1 = 0
global Path2; Path2 = 0
global Path3; Path3 = 0
global Path4; Path4 = 0

# Set the size of the GUI window
root.geometry('{}x{}'.format(800,800))

# Set the initial state of the GUI window
```

```python
root.state('normal')

# Create a frame to hold the top section of the GUI window
Top_frame = ttk.Frame(root)
Top_frame.pack(fill=tk.Y, side="top")

# Create a frame to hold the middle section of the GUI window
Middle_frame = ttk.Frame(root)
Middle_frame.pack(fill=tk.Y, side="top")

# Create a frame to hold the bottom section of the GUI window
Bottom_frame = ttk.Frame(root)
Bottom_frame.pack(fill=tk.Y, side="top")

# Create a frame to hold the pre-defined positions buttons
preset_frame = ttk.LabelFrame(Middle_frame, text="Pre_defined Positions")
preset_frame.pack(padx=10, pady=10, side="right")

# Create a frame to hold the settings buttons
settings_frame = ttk.LabelFrame(Middle_frame, text="Settings", relief =
tk.RAISED)
settings_frame.pack(padx=10, pady=10, side="right")

# Create a frame to hold the motion buttons
motion_frame = ttk.LabelFrame(Middle_frame, text="Motion")
motion_frame.pack(padx=10, pady=10, side="right")

# Create a frame to hold the save position button
save_frame = ttk.LabelFrame(Middle_frame, text="Save position")
save_frame.pack(padx=10, pady=10, side="left")

# Create a frame to hold the Roll joint controls
roll_frame = ttk.LabelFrame(Top_frame, text="Roll", relief = tk.SUNKEN)
roll_frame.pack(padx=10, pady=10, side="right")

# Create a frame to hold the Pitch joint controls
pitch_frame = ttk.LabelFrame(Top_frame, text="Pitch", relief = tk.SUNKEN)
pitch_frame.pack(padx=10, pady=10, side="right")

# Create a frame to hold the Yaw joint controls
yaw_frame = ttk.LabelFrame(Top_frame, text="Yaw", relief = tk.SUNKEN)
yaw_frame.pack(padx=10, pady=10, side="right")

# Create a frame to hold the Z-axis joint controls
Z_axis_frame = ttk.LabelFrame(Top_frame, text="Z asix")
Z_axis_frame.pack(padx=10, pady=10, side="right")

# Create a frame to hold the X-axis joint controls
```

```python
X_axis_frame = ttk.LabelFrame(Top_frame, text="X axis")
X_axis_frame.pack(padx=10, pady=10, side="right")

# Create a frame to hold the Y-axis joint controls
Y_axis_frame = ttk.LabelFrame(Top_frame, text="Y axis")
Y_axis_frame.pack(padx=10, pady=10, side="right")

# Create a frame to hold the Gripper controls
gripper_frame = ttk.LabelFrame(Middle_frame, text="Gripper")
gripper_frame.pack(padx=10, pady=10, side="left")

style = ttk.Style(root)
style.theme_use("scidsand")

# See included styles
our_themes = root.get_themes()

def changer(theme):
    style.theme_use(theme)

my_menu = tk.Menu(root)
root.config(menu=my_menu)

theme_menu = tk.Menu(my_menu, tearoff=0)
my_menu.add_cascade(label="Themes", menu=theme_menu)

#create Sub menu
for t in our_themes:
    theme_menu.add_command(label=t, command= lambda t=t: changer(t))

is_on = False
# Create Label
my_label = ttk.Label(settings_frame,
    text = "Learning mode Is Off!",
    font = ("Helvetica", 10))

my_label.pack(pady = 5)

# Define our switch function
def switch():
    """Make a switch for a activating and deactivating the leaning mode"""

    global is_on

    # Determine is on or off
    if is_on:
        learning_mode_switch.config(image = off)
        my_label.config(text = "Learning mode is Off")
```

```python
        is_on = False
        learning_mode(0)

    else:
        learning_mode_switch.config(image = on)
        my_label.config(text = "Learning mode is On")
        is_on = True
        learning_mode(1)

def Update_position():
    # Update the Pose lable
    Pose_posistion_label.config(text = str(get_pose().position))
    Pose_rpy_label.config(text = str(get_pose().rpy))

    # Update the Joints lable
    Joints123_label.config(text = """Joint1: {}
Joint2: {}
Joint3: {}""".format(Get_joints()[0], Get_joints()[1], Get_joints()[2]))
    Joints456_label.config(text = """Joint4: {}
Joint5: {}
Joint6: {}""".format(Get_joints()[3], Get_joints()[4], Get_joints()[5]))

# Define Our Images
on = ImageTk.PhotoImage(Image.open(path + "/switch-
on.png").resize((60,60)),Image.ANTIALIAS)
off = ImageTk.PhotoImage(Image.open(path + "/switch-
off.png").resize((60,60)),Image.ANTIALIAS)

# Create A Button
learning_mode_switch = ttk.Button(settings_frame, image = off,
                command = switch)
learning_mode_switch.pack(pady = 20)


def set_max_speed():
    """Takes a value from the user between 0 and 1, where 0 = 0% scaling
factor of the speed,
    and 1 = 100% = full speed"""

    factor = simpledialog.askfloat("Max velocity scaling factor", "Enter a
value between 0 and 1")
    arm.set_max_velocity_scaling_factor(factor)

set_speed = ttk.Button(settings_frame,text="Set Max velocity",
command=set_max_speed)
set_speed.pack(pady = 5)

def request_calibration():
```

```python
    """Request a new calibration"""

    rospy.wait_for_service('/niryo_robot/joints_interface/request_new_calibrat
ion')
    service =
rospy.ServiceProxy('/niryo_robot/joints_interface/request_new_calibration',
Trigger)

    response = service()

def Get_Joints_limits():
    """Getting the limits for each joint
    You can get any joint limits as following:

    Get_Joints_limits().joint_limits[0 - 5].max (float)
    Get_Joints_limits().joint_limits[0 - 5].min (float)
    Get_Joints_limits().joint_limits[0 - 5].name (str)

    Where 0 for (joint 1), and 5 for (joint 6)
    max, min, or name would give the maximum, minimum or name of the indicated
joint"""

    rospy.wait_for_service('/niryo_robot_arm_commander/get_joints_limit')
    service =
rospy.ServiceProxy('/niryo_robot_arm_commander/get_joints_limit',
GetJointLimits)

    response = service()

    return response


def calibration():
    """Give a value of 1 to the service responsible for auto calibration"""

    rospy.wait_for_service('/niryo_robot/joints_interface/calibrate_motors')
    service =
rospy.ServiceProxy('/niryo_robot/joints_interface/calibrate_motors', SetInt)

    request = SetIntRequest()
    request.value = 1

    response = service(request)

    if response.success:
        rospy.loginfo("Service call succeeded")
    else:
        rospy.logerr("Service call failed")
```

```python
# Button for Motors calibration.
cal_Btn = ttk.Button(settings_frame,text="Calibrate motors",
command=calibration)
cal_Btn.pack(pady = 5)

# Button for requesting new calibration.
req_cal_Btn = ttk.Button(settings_frame,text="Request new
calibration",command=request_calibration)
req_cal_Btn.pack(pady = 5)

def Motors_reboot():
    """Reboot all motors"""

    rospy.wait_for_service('/niryo_robot_hardware_interface/reboot_motors')
    service =
rospy.ServiceProxy('/niryo_robot_hardware_interface/reboot_motors', Trigger)

    response = service()

# Button for Motors rebooting.
Reboot_Btn = ttk.Button(settings_frame,text="Motors
Reboot",command=Motors_reboot)
Reboot_Btn.pack(pady = 5)

up_roll_img = ImageTk.PhotoImage(Image.open(path + "/up-roll-
arrow.png").resize((40,40)),Image.ANTIALIAS)
down_roll_img = ImageTk.PhotoImage(Image.open(path + "/down-roll-
arrow.png").resize((40,40)),Image.ANTIALIAS)

def Roll_plus():
    """Increase the value of the roll axis"""
    global Plan

    FK = get_pose()
    rpy = FK.rpy
    p = FK.position
    pose = Pose()

    roll = rpy.roll
    roll += 0.1

    pose.position.x = p.x
    pose.position.y = p.y
    pose.position.z = p.z
```

```python
    pose.orientation.x, pose.orientation.y, pose.orientation.z,
pose.orientation.w = tf.transformations.quaternion_from_euler(roll, rpy.pitch,
rpy.yaw)

    arm.set_goal_tolerance(0.01)
    arm.set_pose_target(pose)
    Plan = arm.plan()

    Update_position()

# To excute the planed rotation.
    arm.execute(Plan,wait=True)

roll_plus = ttk.Button(roll_frame, image=down_roll_img, command=Roll_plus)
roll_plus.pack(pady = 5)

def Roll_neg():
    """decrease the value of the roll axis"""
    global Plan

    FK = get_pose()
    rpy = FK.rpy
    p = FK.position
    pose = Pose()

    roll = rpy.roll
    roll -= 0.1

    pose.position.x = p.x
    pose.position.y = p.y
    pose.position.z = p.z

    pose.orientation.x, pose.orientation.y, pose.orientation.z,
pose.orientation.w = tf.transformations.quaternion_from_euler(roll, rpy.pitch,
rpy.yaw)

    arm.set_goal_tolerance(0.01)
    arm.set_pose_target(pose)
    Plan = arm.plan()

    Update_position()

    # To excute the planed rotation.
    arm.execute(Plan,wait=True)

roll_neg = ttk.Button(roll_frame, image=up_roll_img, command=Roll_neg)
roll_neg.pack(pady = 5)
```

```python
def Pitch_plus():
    """Increase the value of the pitch axis"""
    global Plan

    FK = get_pose()
    rpy = FK.rpy
    p = FK.position
    pose = Pose()

    pitch = rpy.pitch
    pitch += 0.1

    pose.position.x = p.x
    pose.position.y = p.y
    pose.position.z = p.z

    pose.orientation.x, pose.orientation.y, pose.orientation.z,
pose.orientation.w = tf.transformations.quaternion_from_euler(rpy.roll, pitch,
rpy.yaw)

    arm.set_goal_tolerance(0.01)
    arm.set_pose_target(pose)
    Plan = arm.plan()

    Update_position()

# To excute the planed rotation.
    arm.execute(Plan,wait=True)

pitch_plus = ttk.Button(pitch_frame, image=down_roll_img, command=Pitch_plus)
pitch_plus.pack(pady = 5)

def Pitch_neg():
    """decrease the value of the pitch axis"""
    global Plan

    FK = get_pose()
    rpy = FK.rpy
    p = FK.position
    pose = Pose()

    pitch = rpy.pitch
    pitch -= 0.1

    pose.position.x = p.x
    pose.position.y = p.y
    pose.position.z = p.z
```

```python
    pose.orientation.x, pose.orientation.y, pose.orientation.z,
pose.orientation.w = tf.transformations.quaternion_from_euler(rpy.roll, pitch,
rpy.yaw)


    arm.set_goal_tolerance(0.01)
    arm.set_pose_target(pose)
    Plan = arm.plan()


    Update_position()


    # To excute the planed rotation.
    arm.execute(Plan,wait=True)

pitch_neg = ttk.Button(pitch_frame, image=up_roll_img, command=Pitch_neg)
pitch_neg.pack(pady = 5)

right_yaw_img = ImageTk.PhotoImage(Image.open(path + "/right-yaw-
arrow.png").resize((40,40)),Image.ANTIALIAS)
left_yaw_img = ImageTk.PhotoImage(Image.open(path + "/left-yaw-
arrow.png").resize((40,40)),Image.ANTIALIAS)

def Yaw_plus():
    """Increase the value of Yaw axis"""
    global Plan

    FK = get_pose()
    rpy = FK.rpy
    p = FK.position
    pose = Pose()

    yaw = rpy.yaw
    yaw += 0.1

    pose.position.x = p.x
    pose.position.y = p.y
    pose.position.z = p.z

    pose.orientation.x, pose.orientation.y, pose.orientation.z,
pose.orientation.w = tf.transformations.quaternion_from_euler(rpy.roll,
rpy.pitch, yaw)

    arm.set_goal_tolerance(0.01)
    arm.set_pose_target(pose)
    Plan = arm.plan()

    Update_position()

# To excute the planed rotation.
```

```python
    arm.execute(Plan,wait=True)

def Yaw_neg():
    """Decrease the value of Yaw axis"""
    global Plan

    FK = get_pose()
    rpy = FK.rpy
    p = FK.position
    pose = Pose()

    yaw = rpy.yaw
    yaw -= 0.1

    pose.position.x = p.x
    pose.position.y = p.y
    pose.position.z = p.z

    pose.orientation.x, pose.orientation.y, pose.orientation.z,
pose.orientation.w = tf.transformations.quaternion_from_euler(rpy.roll,
rpy.pitch, yaw)

    arm.set_goal_tolerance(0.01)
    arm.set_pose_target(pose)
    Plan = arm.plan()

    Update_position()

    # To excute the planed rotation.
    arm.execute(Plan,wait=True)

yaw_plus = ttk.Button(yaw_frame, image=left_yaw_img, command=Yaw_neg)
yaw_plus.pack(pady = 5, side="right")

yaw_neg = ttk.Button(yaw_frame, image=right_yaw_img, command=Yaw_plus)
yaw_neg.pack(pady = 5, side="left")

def Up_z():
    """Increase the value of Z-axis"""
    global Plan

    FK = get_pose()
    rpy = FK.rpy
    p = FK.position
    pose = Pose()

    z = p.z
    z += 0.02
```

```python
    pose.position.x = p.x
    pose.position.y = p.y
    pose.position.z = z

    pose.orientation.x, pose.orientation.y, pose.orientation.z,
pose.orientation.w = tf.transformations.quaternion_from_euler(rpy.roll,
rpy.pitch, rpy.yaw)

    arm.set_goal_tolerance(0.01)
    arm.set_pose_target(pose)
    Plan = arm.plan()

    Update_position()

# To excute the planed rotation.
    arm.execute(Plan,wait=True)

up_img = ImageTk.PhotoImage(Image.open(path + "/up-
arrow.png").resize((30,30)),Image.ANTIALIAS)
down_img = ImageTk.PhotoImage(Image.open(path + "/down-
arrow.png").resize((30,30)),Image.ANTIALIAS)

up_z = ttk.Button(Z_axis_frame, image=up_img, command=Up_z)
up_z.pack(pady = 5)

def Down_z():
    """Decrease the value of Z-axis"""
    global Plan

    FK = get_pose()
    rpy = FK.rpy
    p = FK.position
    pose = Pose()

    z = p.z
    z -= 0.02

    pose.position.x = p.x
    pose.position.y = p.y
    pose.position.z = z

    pose.orientation.x, pose.orientation.y, pose.orientation.z,
pose.orientation.w = tf.transformations.quaternion_from_euler(rpy.roll,
rpy.pitch, rpy.yaw)

    arm.set_goal_tolerance(0.01)
    arm.set_pose_target(pose)
```

```python
    Plan = arm.plan()

    Update_position()

    # To excute the planed rotation.
    arm.execute(Plan,wait=True)

down_z = ttk.Button(Z_axis_frame, image=down_img, command=Down_z)
down_z.pack(pady = 5)

right_img = ImageTk.PhotoImage(Image.open(path + "/right-
arrow.png").resize((30,30)),Image.ANTIALIAS)
left_img = ImageTk.PhotoImage(Image.open(path + "/left-
arrow.png").resize((30,30)),Image.ANTIALIAS)

def Right_x():
    """Increase the value of X-axis"""
    global Plan

    FK = get_pose()
    rpy = FK.rpy
    p = FK.position
    pose = Pose()

    x = p.x
    x += 0.02

    pose.position.x = x
    pose.position.y = p.y
    pose.position.z = p.z

    pose.orientation.x, pose.orientation.y, pose.orientation.z,
pose.orientation.w = tf.transformations.quaternion_from_euler(rpy.roll,
rpy.pitch, rpy.yaw)

    arm.set_goal_tolerance(0.01)
    arm.set_pose_target(pose)
    Plan = arm.plan()

    Update_position()

# To excute the planed rotation.
    arm.execute(Plan,wait=True)

right = ttk.Button(X_axis_frame, image=right_img, command=Right_x)
right.pack(pady = 5, side="right")

def Left_x():
```

```python
    """Decrease the value of X-axis"""
    global Plan

    FK = get_pose()
    rpy = FK.rpy
    p = FK.position
    pose = Pose()

    x = p.x
    x -= 0.02

    pose.position.x = x
    pose.position.y = p.y
    pose.position.z = p.z

    pose.orientation.x, pose.orientation.y, pose.orientation.z,
pose.orientation.w = tf.transformations.quaternion_from_euler(rpy.roll,
rpy.pitch, rpy.yaw)

    arm.set_goal_tolerance(0.01)
    arm.set_pose_target(pose)
    Plan = arm.plan()

    Update_position()

    # To excute the planed rotation.
    arm.execute(Plan,wait=True)

left = ttk.Button(X_axis_frame, image=left_img, command=Left_x)
left.pack(pady = 5, side="left")

def Up_y():
    """Increase the value of Y-axis"""
    global Plan

    FK = get_pose()
    rpy = FK.rpy
    p = FK.position
    pose = Pose()

    y = p.y
    y += 0.02

    pose.position.x = p.x
    pose.position.y = y
    pose.position.z = p.z
```

```python
    pose.orientation.x, pose.orientation.y, pose.orientation.z,
pose.orientation.w = tf.transformations.quaternion_from_euler(rpy.roll,
rpy.pitch, rpy.yaw)

    arm.set_goal_tolerance(0.01)
    arm.set_pose_target(pose)
    Plan = arm.plan()

    Update_position()

# To excute the planed rotation.
    arm.execute(Plan,wait=True)

up_y = ttk.Button(Y_axis_frame, image=up_img, command=Up_y)
up_y.pack(pady = 5)

def Down_y():
    """Decrease the value of Y-axis"""
    global Plan

    FK = get_pose()
    rpy = FK.rpy
    p = FK.position
    pose = Pose()

    y = p.y
    y -= 0.02

    pose.position.x = p.x
    pose.position.y = y
    pose.position.z = p.z

    pose.orientation.x, pose.orientation.y, pose.orientation.z,
pose.orientation.w = tf.transformations.quaternion_from_euler(rpy.roll,
rpy.pitch, rpy.yaw)

    arm.set_goal_tolerance(0.01)
    arm.set_pose_target(pose)
    Plan = arm.plan()

    Update_position()

    # To excute the planed rotation.
    arm.execute(Plan,wait=True)

down_y = ttk.Button(Y_axis_frame, image=down_img, command=Down_y)
down_y.pack(pady = 5)
```

```python
def open_gripper():
    """Using the open gripper service to open the end effetor (gripper)
    with ID = 11, and the open position = 2060."""

    rospy.wait_for_service('/niryo_robot/tools/open_gripper')
    service = rospy.ServiceProxy('/niryo_robot/tools/open_gripper',
ToolCommand)

    request = ToolCommandRequest()
    request.id = 11
    request.position = 2060
    request.speed = 500
    request.hold_torque = 50
    request.max_torque = 100

    response = service(request)

# Button for opening the gripper.
open_ = ttk.Button(gripper_frame,text="open griper", command=open_gripper)
open_.pack(pady=5)

def close_gripper():
    """Using the close gripper service to close the end effetor (gripper)
    with ID = 11, and the closed position = 807."""

    rospy.wait_for_service('/niryo_robot/tools/close_gripper')
    service = rospy.ServiceProxy('/niryo_robot/tools/close_gripper',
ToolCommand)

    request = ToolCommandRequest()
    request.id = 11
    request.position = 807
    request.speed = 500
    request.hold_torque = 100
    request.max_torque = 100

    response = service(request)

# Button for closing the gripper.
close_ = ttk.Button(gripper_frame,text="close griper", command=close_gripper)
close_.pack(pady=5)

global all_joints; all_joints = []
def learning_mode(on_or_off=0):
    """The function takes argument 0 or 1, whereas, 1 = learning more is
turned on,
    and 0 = learning mode is off."""
```

```python
    rospy.wait_for_service('/niryo_robot/learning_mode/activate')
    service = rospy.ServiceProxy('/niryo_robot/learning_mode/activate',
SetBool)

    request = SetBoolRequest()
    global all_joints

    if on_or_off == 1:
        request.value = True
    elif on_or_off == 0:
        request.value = False

        joints = (Get_joints())
        all_joints.append(joints)
        with open('joints_values', 'w') as joints_values:
            csv_writer = csv.writer(joints_values)

            for line in all_joints:
                csv_writer.writerow(line)


    Update_position()

    response = service(request)

def Get_joints():
    """subscribe to the topic /Joint_states and take the joints' values"""
    msg = rospy.wait_for_message('/joint_states', JointState)
    joints = msg.position

    # return a tuple of 6 value for each joint from 1 till 6.
    return joints

def get_pose():
    """
    @argm: None

    return a RobotState message type, like the following:
position:
  x: ------
  y: ------
  z: ------
rpy:
  roll: ------
  pitch: ------
  yaw: ------
orientation:
  x: ------
```

```
  y: ------
  z: ------
  w: ------
twist:
  linear:
    x: 0.0
    y: 0.0
    z: 0.0
  angular:
    x: 0.0
    y: -0.0
    z: -0.0
tcp_speed: 0.0
    """

    msg = rospy.wait_for_message('/niryo_robot/robot_state', RobotState)

    return msg


def get_pose_list():
    """
    return the current pose as a list:
    [x, y, z, roll, pitch, yaw]
    """

    value = get_pose()
    p = value.position
    rpy = value.rpy

    return [p.x, p.y, p.z, rpy.roll, rpy.pitch, rpy.yaw]

# Generate a lable for Pose positions in the bottom frame.
Pose_posission_label = ttk.Label(Bottom_frame,
    text = str(get_pose().position),
    font = ("Helvetica", 12))
Pose_posission_label.pack(padx=10, side="left")

# Generate a lable for Pose rpy in the bottom frame.
Pose_rpy_label = ttk.Label(Bottom_frame,
    text = str(get_pose().rpy),
    font = ("Helvetica", 12))
Pose_rpy_label.pack(padx=10, side="left")

# Generate a lable for Joints in the bottom frame.
Joints123_label = ttk.Label(Bottom_frame,
    text = """Joint1: {}
Joint2: {}
```

```python
Joint3: {}""".format(round(Get_joints()[0], 6), round(Get_joints()[1], 6),
round(Get_joints()[2], 6)),
    font = ("Helvetica", 12))
Joints123_label.pack(padx=10, side="left")

Joints456_label = ttk.Label(Bottom_frame,
    text = """Joint4: {}
Joint5: {}
Joint6: {}""".format(round(Get_joints()[3], 6), round(Get_joints()[4], 6),
round(Get_joints()[5], 6)),
    font = ("Helvetica", 12))
Joints456_label.pack(padx=10, side="left")


def Print_current_pose():
    FK = get_pose()
    print(FK.position)
    print('\n')
    print(FK.rpy)

PrintFK = ttk.Button(settings_frame,text="Print current FK",
command=Print_current_pose)
PrintFK.pack(pady=5)

def rotate():
    global Plan

    FK = get_pose()
    rpy = FK.rpy
    p = FK.position
    pose = Pose()

    pose.position.x = p.x
    pose.position.y = p.y
    pose.position.z = p.z
    roll = rpy.roll
    pitch = rpy.pitch
    yaw = rpy.yaw

    for i in range(10):
        if roll >= -2.5:
            roll -= 0.1
        else:
            pass

    pose.orientation.x, pose.orientation.y, pose.orientation.z,
pose.orientation.w = tf.transformations.quaternion_from_euler(roll, rpy.pitch,
rpy.yaw)
```

```python
    arm.set_goal_tolerance(0.01)
    arm.set_pose_target(pose)
    Plan = arm.plan()

    Update_position()

    # To excute the planed rotation.
    arm.execute(Plan,wait=True)

#Rotate = ttk.Button(gripper_frame,text="Rotate", command=rotate)
#Rotate.pack()
global Plan_Joints_withspeed
Plan_Joints_withspeed = (St-Onge & Herath, 2022)
def save_all_fo():
    global Plan_Joints_withspeed

    #arm_variable_values = Get_joints()
    Plan_Joints_withspeed["values"].append(Get_joints())

    Plan_Joints_withspeed["speed"].append(1)
    Plan_Joints_withspeed["gripper"].append(1)


def save_all_fc():
    global Plan_Joints_withspeed

    #arm_variable_values = Get_joints()
    Plan_Joints_withspeed["values"].append(Get_joints())

    Plan_Joints_withspeed["speed"].append(1)
    Plan_Joints_withspeed["gripper"].append(0)


def save_all_so():
    global Plan_Joints_withspeed

    #arm_variable_values = Get_joints()
    Plan_Joints_withspeed["values"].append(Get_joints())

    Plan_Joints_withspeed["speed"].append(0.3)
    Plan_Joints_withspeed["gripper"].append(1)


def save_all_sc():
    global Plan_Joints_withspeed

    #arm_variable_values = Get_joints()
    Plan_Joints_withspeed["values"].append(Get_joints())

    Plan_Joints_withspeed["speed"].append(0.3)
```

```python
        Plan_Joints_withspeed["gripper"].append(0)

Save_Fast_open = ttk.Button(gripper_frame,text="Save Fast & OPEN",
command=save_all_fo)
Save_Fast_open.pack()

Save_Fast_close = ttk.Button(gripper_frame,text="Save Fast & CLOSE",
command=save_all_fc)
Save_Fast_close.pack()

Save_Slow_open = ttk.Button(gripper_frame,text="Save Slow & OPEN",
command=save_all_so)
Save_Slow_open.pack()

Save_Slow_close = ttk.Button(gripper_frame,text="Save Slow & CLOSE",
command=save_all_sc)
Save_Slow_close.pack()

def reset_saved_paths():
    global Plan_Joints_withspeed
    Plan_Joints = []
    Plan_Joints_withspeed = {"values": [], "speed": [], "gripper": []}

Reset_saved_paths = ttk.Button(gripper_frame,text="Reset saved paths",
command=reset_saved_paths)
Reset_saved_paths.pack()

def excute_all():
    global Plan_Joints_withspeed
    m = simpledialog.askinteger("How many time?", "Enter a value from 1 till
10")

    for n in range(m):
        for i in range(len(Plan_Joints_withspeed["speed"])):
            arm.set_max_velocity_scaling_factor(Plan_Joints_withspeed["speed"]
[i])
            arm.set_joint_value_target(Plan_Joints_withspeed["values"][i])
            arm.execute(arm.plan(),wait=True)
            if Plan_Joints_withspeed["gripper"][i]==1:
                open_gripper()
            elif Plan_Joints_withspeed["gripper"][i]==0:
                close_gripper()
            else:
                pass

excute_saved_values = ttk.Button(gripper_frame,text="Excute saved values",
command=excute_all)
excute_saved_values.pack()
```

```python
def Save_Pose_Path1():
    """Takes the current Pose values and assign it to the Path1 variable"""

    global Path1
    FK = get_pose()
    rpy = FK.rpy
    p = FK.position
    pose = Pose()

    pose.position.x = p.x
    pose.position.y = p.y
    pose.position.z = p.z

    pose.orientation.x, pose.orientation.y, pose.orientation.z,
pose.orientation.w = tf.transformations.quaternion_from_euler(rpy.roll,
rpy.pitch, rpy.yaw)

    Path1 = pose

save_pose1 = ttk.Button(save_frame,text="Save Position Path1",
command=Save_Pose_Path1)
save_pose1.pack(pady=5)

def Plan_Saved_Path1_Click():
    """Plan Path1"""

    global Plan
    global Path1

    if Path1:
        pose = Path1
        arm.set_goal_tolerance(0.001)
        arm.set_pose_target(pose)
        Plan = arm.plan()
    else:
        pass

Plan_Path1 = ttk.Button(save_frame,text="Plan Path1",
command=Plan_Saved_Path1_Click)
Plan_Path1.pack(pady=5)

def Save_Pose_Path2():
    """Takes the current Pose values and assign it to the Path2 variable"""

    global Path2
    FK = get_pose()
    rpy = FK.rpy
```

```python
    p = FK.position
    pose = Pose()

    pose.position.x = p.x
    pose.position.y = p.y
    pose.position.z = p.z

    pose.orientation.x, pose.orientation.y, pose.orientation.z,
pose.orientation.w = tf.transformations.quaternion_from_euler(rpy.roll,
rpy.pitch, rpy.yaw)

    Path2 = pose

save_pose2 = ttk.Button(save_frame,text="Save Position Path2",
command=Save_Pose_Path2)
save_pose2.pack(pady=5)

def Plan_Saved_Path2_Click():
    """Plan path2"""

    global Plan
    global Path2

    if Path2:
        pose = Path2
        arm.set_goal_tolerance(0.001)
        arm.set_pose_target(pose)
        Plan = arm.plan()
    else:
        pass

Plan_Path2 = ttk.Button(save_frame,text="Plan Path2",
command=Plan_Saved_Path2_Click)
Plan_Path2.pack(pady=5)

def Save_Pose_Path3():
    """Takes the current Pose values and assign it to the Path3 variable"""

    global Path3
    FK = get_pose()
    rpy = FK.rpy
    p = FK.position
    pose = Pose()

    pose.position.x = p.x
    pose.position.y = p.y
    pose.position.z = p.z
```

```python
    pose.orientation.x, pose.orientation.y, pose.orientation.z,
pose.orientation.w = tf.transformations.quaternion_from_euler(rpy.roll,
rpy.pitch, rpy.yaw)


    Path3 = pose

save_pose3 = ttk.Button(save_frame,text="Save Position Path3",
command=Save_Pose_Path3)
save_pose3.pack(pady=5)

def Plan_Saved_Path3_Click():
    """Plan path3"""

    global Plan
    global Path3

    if Path3:
        pose = Path3
        arm.set_goal_tolerance(0.001)
        arm.set_pose_target(pose)
        Plan = arm.plan()
    else:
        pass

Plan_Path3 = ttk.Button(save_frame,text="Plan Path3",
command=Plan_Saved_Path3_Click)
Plan_Path3.pack(pady=5)

def Save_Pose_Path4():
    """Takes the current Pose values and assign it to the Path4 variable"""

    global Path4
    FK = get_pose()
    rpy = FK.rpy
    p = FK.position
    pose = Pose()

    pose.position.x = p.x
    pose.position.y = p.y
    pose.position.z = p.z

    pose.orientation.x, pose.orientation.y, pose.orientation.z,
pose.orientation.w = tf.transformations.quaternion_from_euler(rpy.roll,
rpy.pitch, rpy.yaw)


    Path4 = pose
```

```python
save_pose4 = ttk.Button(save_frame,text="Save Position Path4",
command=Save_Pose_Path4)
save_pose4.pack(pady=5)

def Plan_Saved_Path4_Click():
    """Plan path4"""

    global Plan
    global Path4

    if Path4:
        pose = Path4
        arm.set_goal_tolerance(0.001)
        arm.set_pose_target(pose)
        Plan = arm.plan()
    else:
        pass

Plan_Path4 = ttk.Button(save_frame,text="Plan Path4",
command=Plan_Saved_Path4_Click)
Plan_Path4.pack(pady=5)

def Play_all_saved_paths():
    """Plan and excute all saved paths sequentially for n number of cycles,
    where n is an integer given by the user between 1 and 10"""

    global Plan
    global Path1
    global Path2
    global Path3
    global Path4

    n = simpledialog.askinteger("How many time?", "Enter a value from 1 till
10")

    for i in range(n):
        if Path1:
            Plan_Saved_Path1_Click()
            ExecutebtnClick()
        if Path2:
            Plan_Saved_Path2_Click()
            ExecutebtnClick()
        if Path3:
            Plan_Saved_Path3_Click()
            ExecutebtnClick()
        if Path4:
            Plan_Saved_Path4_Click()
            ExecutebtnClick()
```

```python
Play_all = ttk.Button(save_frame,text="Play all saved paths",
command=Play_all_saved_paths)
Play_all.pack(pady=5)

def GoHomebtnClick():
    """Plan the the pre-saved position called resting"""

    global Plan
    arm.set_goal_tolerance(0.01)
    arm.set_named_target("resting")
    Plan = arm.plan()
    #arm.go()

GoHomebtn = ttk.Button(preset_frame,text="Go to resting position",width=20,
command=GoHomebtnClick)
GoHomebtn.pack(pady=5)

def GoUpbtnClick():
    """Plan the the pre-saved position called straight_up"""

    global Plan
    arm.set_goal_tolerance(0.01)
    arm.set_named_target("straight_up")
    Plan = arm.plan()

GoUpbtn = ttk.Button(preset_frame,text="Go to Up position",width=20,
command=GoUpbtnClick)
GoUpbtn.pack(pady=5)

def GoForwardbtnClick():
    """Plan the the pre-saved position called straight_forward"""

    global Plan
    arm.set_goal_tolerance(0.01)
    arm.set_named_target("straight_forward")
    Plan = arm.plan()

GoForwardbtn = ttk.Button(preset_frame,text="Go to Forward position",width=20,
command=GoForwardbtnClick)
GoForwardbtn.pack(pady=5)

def Plan1btnClick():
    """Plan an path with a given pose coordinations"""
    global Plan

    pose = Pose()
    pose.position.x = 0.035
```

```python
    pose.position.y = -0.432
    pose.position.z = 0.134
    pose.orientation.x, pose.orientation.y, pose.orientation.z,
pose.orientation.w = tf.transformations.quaternion_from_euler(-0.458706689443,
0.477070781868, -1.24515102216)

    arm.set_goal_tolerance(0.01)
    arm.set_pose_target(pose)
    Plan = arm.plan()

Plan1btn = ttk.Button(preset_frame,text="Plan Path place",width=20,
command=Plan1btnClick)
Plan1btn.pack(pady=5)

def random():
    """Plan a random position"""
    global Plan
    arm.set_random_target()
    Plan = arm.plan()

# Dangrous!!!!!!
#randombtn = ttk.Button(preset_frame,text="Plan a random path",width=20,
command=random)
#randombtn.pack(pady=5)

def ExecutebtnClick():
    """Execute the planed path in the global variable Plan"""

    global Plan
    arm.execute(Plan,wait=True)

    Update_position()

Executebtn = ttk.Button(preset_frame,text="Execute",width=20,
command=ExecutebtnClick)
Executebtn.pack(pady=5)

#Stopbtn = ttk.Button(settings_frame,text="Stop", command=arm.stop)
#Stopbtn.pack(pady=5)

root.mainloop()
```